Server-side Log Data Analytics for I/O Workload Characterization and Coordination on Large Shared Storage Systems

Yang Liu[⋆], Raghul Gunasekaran[†], Xiaosong Ma[⋄], and Sudharshan S. Vazhkudai[†]

[⋆]*North Carolina State University, yliu43@ncsu.edu*
[⋄]*Qatar Computing Research Institute, Hamad Bin Khalifa University, xma@qf.org.qa*
[†]*Oak Ridge National Laboratory, {gunasekaranr, vazhkudaiss}@ornl.gov*

***Abstract*—Inter-application I/O contention and performance interference have been recognized as severe problems. In this work, we demonstrate, through measurement from Titan (world's No. 3 supercomputer), that high I/O variance co-exists with the fact that individual storage units remain under-utilized for the majority of the time. This motivates us to propose AID, a system that performs automatic application I/O characterization and I/O-aware job scheduling. AID analyzes *existing* I/O traffic and batch job history logs, without any prior knowledge on applications or user/developer involvement. It identifies the small set of I/O-intensive candidates among all applications running on a supercomputer and subsequently mines their I/O patterns, using more detailed per-I/O-node traffic logs. Based on such auto-extracted information, AID provides online I/O-aware scheduling recommendations to steer I/O-intensive applications away from heavy ongoing I/O activities.**

We evaluate AID on Titan, using both real applications (with extracted I/O patterns validated by contacting users) and our own pseudo-applications. Our results confirm that AID is able to (1) identify I/O-intensive applications and their detailed I/O characteristics, and (2) significantly reduce these applications' I/O performance degradation/variance by jointly evaluating outstanding applications' I/O pattern and real-time system I/O load.

I. Introduction

HPC facilities support multiple concurrently executing workloads with shared storage. For instance, the center-wide Lustre-based parallel file system, Spider [26], at Oak Ridge National Laboratory (ORNL) provides 30PB of capacity and over 1TB/s aggregate I/O throughput, serving several machines including Titan, the current No. 3 supercomputer [3].

Like most large shared resources, HPC storage systems over-provision I/O bandwidth. On average, individual pieces of hardware (such as I/O server nodes and disks) are often under-utilized. Figure 1 illustrates this with the cumulative distribution of I/O throughput on each of Spider's Lustre OSTs (Object Storage Targets) during a 5-month period in 2015, giving the percentage of time each individual OST spends at different throughput levels. Overall, most of the OSTs are not busy, experiencing less than 1% (5MB/s) and 20% (100MB/s) of their individual peak throughput during 88.4% and 99.6% of system time, respectively.

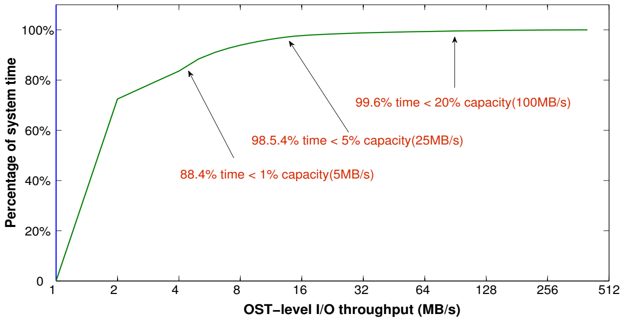


Fig. 1. CDF of per-OST I/O throughput

Even so, I/O-heavy jobs may collide, creating contention-induced performance variance, a recognized challenge for I/O-related performance debugging and optimization [14], [15], [19], [27], [29]. Furthermore, as hard disks remain the dominant media, I/O contention leads to excessive seeks, degrading the overall I/O throughput.

One major reason for such I/O-induced performance variance is the I/O-oblivious job scheduling: supercomputer jobs are typically dispatched in a FIFO order plus backfilling, with further priority-based adjustment [2]. While there are several studies aimed at reducing inter-job I/O contention [28], [39], *I/O-aware job scheduling has never been available on production HPC systems*. The major obstacle lies in the cost to obtain per-application parallel I/O characteristics through tracing/profiling and the difficulty for a supercomputer to demand such information from users/developers.

In this work, we propose AID (Automatic I/O Diverter), an I/O-aware job scheduling mechanism built on the zero-overhead hardware monitoring already available on supercomputer storage servers [24]. AID correlates the coarse-grained server-side I/O traffic log (aggregate and OST-level) to (1) identify I/O-intensive applications, (2) "mine" the I/O traffic pattern of applications classified as I/O-intensive, and (3) provide job scheduling suggestions on whether an I/O-

SC16; Salt Lake City, Utah, USA; November 2016
978-1-4673-8815-3/16/$31.00 ©2016 IEEE

intensive job can be immediately dispatched.

Note that AID achieves the above goals fully automatically, *without requiring any apriori information on the applications or jobs, additional tracing/profiling, or effort from developers/users*. This is important as typical supercomputers cannot achieve universal participation from users for collecting per-job I/O patterns. Instead, AID transparently examines the full job log and identifies common I/O patterns across multiple executions of the same application. This is based on the same intuition exploited by our prior work leveraging supercomputer I/O traffic logs [18]: the common behavior observed across multiple executions of the same application is likely attributed to this application. However, our prior work identifies the *I/O signature* of a *given I/O-intensive application* and makes the strong assumption that its job run instances are identical executions. In contrast, AID takes a fully data-driven approach, sifting out dozens of I/O-heavy applications from job-I/O history containing millions of jobs running thousands of unique applications. These applications can then be given special attention in scheduling to reduce I/O contention. Also, AID is able to tolerate common behavior variances in an application's repeated execution (such as running for varied number of timesteps). Finally, AID utilizes detailed per-OST logs that became available more recently, while our prior work only studies aggregate traffic.

We have implemented AID and evaluated its effectiveness in I/O characterization on Titan, using real applications (partially validated by querying their users), as well as pseudo-applications (where "ground truth" is readily available). For validated I/O-intensive applications, we verified the accuracy of AID's I/O pattern identification. Finally, we assessed the potential gain of I/O-aware job scheduling. Our results confirm that AID can successfully identify I/O-intensive applications and their high-level I/O characteristics. While we currently do not have the means to deploy new scheduling policies on Titan, our proof-of-concept evaluation indicates that I/O-aware scheduling might be highly promising for future systems.

II. Background

Our work targets petascale or larger platforms. Below we present an overview of one such storage system, the ORNL Spider file system [26], supporting Titan and several other clusters. It is also where we obtain log data, and perform experimental evaluation.

A. Spider Storage Infrastructure

Figure 2 shows the Spider architecture, running Lustre [8]. Spider's 20,160 SATA drives are managed by 36 DDN SFA12K RAID controllers (henceforth referred to as *controllers*). Every 10 disks form a RAID 6 array that makes a Lustre Object Storage Target (OST). Access is via the 288 Lustre Object Storage Servers (OSSes), each with 7 OSTs attached, partitioned into two independent and non-overlapping namespaces, *atlas1* and *atlas2*, for load-balancing and capacity management. Each partition includes half of the system, 144 Lustre OSSes and 1,008 OSTs. The compute nodes (clients) connect to Spider over a multistage InfiniBand network (SION).

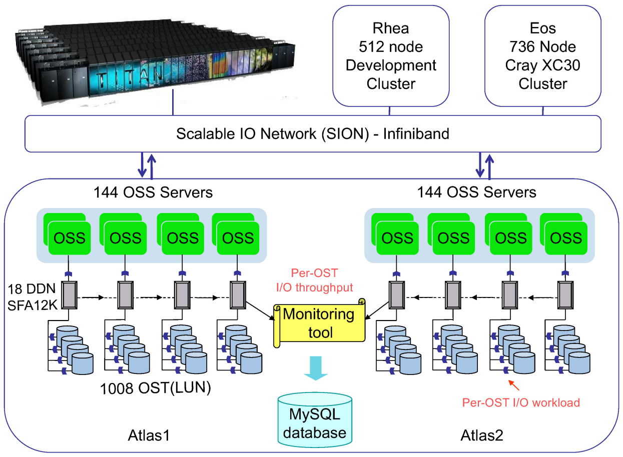


Fig. 2. Spider architecture

B. I/O and Job Data Collection

I/O traffic logs Server-side I/O statistics have been collected on Spider since 2009, via a custom API provided by the DDN controllers. A custom daemon utility [24] polls the controllers periodically and stores the results in a MySQL database. Data collected include read/write I/O throughput and IOPS, I/O request sizes, etc., amounting to around 4GB of log data per day. Unlike client-side or server-side I/O tracing, such coarse-grained monitoring/logging via the separate management Ethernet network has negligible overhead.

Applications are allocated a set of OSTs. Based on the *stripe width* k, the I/O client round robins across the k $OSTs$. The current Spider system has monitoring tools capturing *per-OST* I/O activity. In this work, we leverage such OST-level information for I/O pattern mining and scheduling.

Batch job logs Most supercomputers maintain a *batch job log*, recording information items like the user/project ID, number of nodes used, job submission, start/end times, job name/ID, etc. By juxtaposing the I/O traffic and job logs, one may *mine* the correlation between I/O traffic and applications' (repeated) executions, to obtain information on application I/O patterns in a lightweight and non-intrusive manner.

III. Approach Overview

A. Problem Definition

Our work has two goals: (1) to *automatically* identify I/O-intensive applications and their I/O access patterns, and (2) to explore I/O-aware job scheduling that staggers jobs running such applications identified as I/O-intensive.

While I/O-aware job scheduling is the ultimate goal for approaching the inter-job I/O interference problem, to deploy batch scheduler modifications on a large production system is beyond the scope of this paper. Meanwhile, there lacks mature parallel file system simulators and it is very hard to generate a realistic background workload mixture. Therefore, rather than developing an enhanced scheduler, we assess the potential benefit of our proposed I/O-aware scheduling by making simple recommendations ("now" or "later") considering application I/O pattern and current system load. Validation is done by comparing the results of I/O-intensive job execution under different recommendations.

The input to AID will be the batch job and server-side I/O traffic logs covering a common time period. AID mines the two logs jointly to identify a set of I/O-intensive applications. Such mining is done continuously and incrementally, with new log data appended daily. For each application labeled as I/O-intensive, AID further identifies its major I/O characteristics. Its job scheduling is then augmented by such I/O-aware analysis, taking into account I/O characteristics (plus the current system I/O load as additional input). The scheduling output is a simple recommendation, in the form of "run" (to dispatch the job in question now) or "delay" (to hold the job in queue and re-examine at the next event-driven scheduling point, such as upon an active job's completion).

B. *Challenges and Overall Design*

This work shares several common building blocks with the IOSI tool [18]. For a given I/O-intensive application, IOSI automatically extracts its *I/O-signature*, plotting the application's I/O throughput along its execution timeline. Common to both tools is the fact that the feasibility of automatic application I/O characterization is established based on *periodic* and *bursty* I/O patterns of HPC applications [34], with the same application executed *repetitively* through many batch jobs [9]. The bursty behavior creates *I/O bursts*, phases of elevated I/O activity, identified from the background noise using techniques such as Wavelet Transform and throughput level filtering [18]. I/O burst serves as the basic unit of per-application I/O traffic identification. The periodic behavior establishes a consistent "pattern", facilitating the attribution of I/O traffic to specific applications. The repetitive behavior allows pattern identification by further correlating multiple *samples* (segments of I/O traffic logs intercepted by the job start/end times of the target application), identifying the commonality across samples as application-affiliated "signal" and difference as "noise."

However, IOSI has I/O-intensive applications *pre-identified* and samples from *guaranteed identical job runs* (same executable and input). AID's focus, in contrast, is to identify "suspected I/O-intensive applications", a fraction of the thousands of unique applications generating millions of batch jobs, *without any I/O-related information about them*. Such automatic classification is quite challenging for several reasons.

First, we cannot assume that supercomputer jobs running the same application are identical. For example, 20 runs during 11/2014-01/2015 of one application had execution times ranging from 1958 to 86644 seconds. While I/O-intensive applications do tend to possess common periodic I/O patterns, large time variance makes sample alignment and I/O burst identification harder, especially without apriori knowledge on an application's I/O intensity.

Second, in some cases application runs do contain *inconsistent I/O patterns*. As periodic I/O itself is a controllable operation, non-production runs checking algorithmic correctness or tuning computation performance often have I/O off or at reduced frequency. While there are applications that seldom change such configurations, the existence of a change in I/O pattern is more challenging to identify than the execution time variance, and further complicates our classification.

Considering these challenges, we focus on applications/jobs with heavy I/O demands, which compose a small fraction of HPC applications. A recent study using Argonne's Darshan I/O profiling tool [21] observed that the aggregate throughput for 75% of applications never exceeded 1GB/s, roughly 1% of the peak platform bandwidth. Our server-side, black-box analysis shares similar observation with this application-side, white-box investigation.

Here we consider an application *I/O-intensive*, if we can identify consistent I/O bursts across its multiple runs, without adopting any specific threshold. The intuition is that I/O-intensive parallel applications do possess intensity (per-application average throughput of 10-200GB/s among the AID-identified intensive applications) and certain kind of patterns to be picked up by AID.

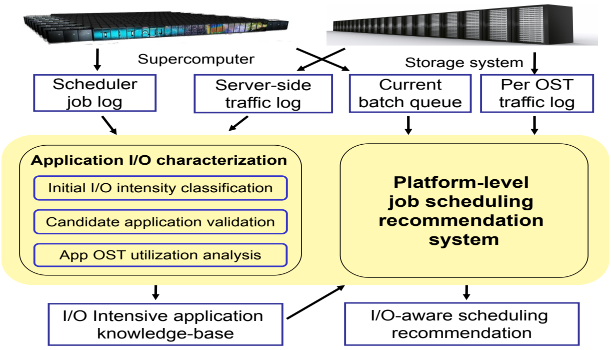

Supercomputer
Storage system
Scheduler job log
Server-side traffic log
Current batch queue
Per OST traffic log
Application I/O characterization
Initial I/O intensity classification
Candidate application validation
App OST utilization analysis
Platform-level job scheduling recommendation system
I/O Intensive application knowledge-base
I/O-aware scheduling recommendation

Fig. 3. AID software architecture

Figure 3 outlines AID's structure, which comprises two major components: (1) an offline application I/O characterization engine that incrementally processes the I/O traffic logs and batch job logs, saving analysis results in an application knowledge base, and (2) an online I/O-aware scheduler that queries the knowledge base and real-time system load levels to make scheduling recommendations for identified I/O-intensive applications. Unlike IOSI, AID also utilizes the more recently available per-OST I/O traffic log to analyze the number of OSTs used simultaneously by the application.

IV. Application I/O Characterization

AID's I/O characterization relies solely on its two input datasets: the job log and the I/O traffic log. The I/O traffic log comes in at per-OST granularity, which provides valuable additional insight to split I/O activities from concurrent jobs. There is little return, however, for the huge time required to examine the full combination of thousands of OSTs and millions of jobs without knowing *anything* about *any* of the applications. Further perplexing the situation, many applications create new files during each I/O burst. E.g., an application may aggregate its output from 2048 processes to 64 shared files every 1000 computation timesteps. Though its *OST footprint* (number of OSTs used during each I/O burst) is stable, each wave of files are assigned and striped to their destination OSTs at file creation time. Therefore, the subset of OSTs showing I/O traffic from this application *migrates* between I/O bursts.

Considering the above, AID takes a two-phase strategy, to first look at the aggregate traffic log and identify a set of applications suspected to be I/O-intensive, which we call *I/O-intensive application candidates* (*candidates* in short for the rest of the paper). The intuition is that if an application is I/O-intensive enough (having recognizable I/O bursts–see details in Sec III-B) and "important" enough (running long or frequently), we are confident enough to mark it as a candidate. Therefore, instead of setting an arbitrary quantitative standard for an application, which should vary with system configurations and load levels, here "being I/O-intensive" is defined as "having *recognizable* and *periodic* I/O pattern". Note that most resource-intensive applications we have seen running on supercomputer perform periodic I/O, so AID focuses on characterizing such applications, while its I/O-aware scheduling might help the minority applications having very irregular I/O patterns as well.

With the short-listed candidates, the second phase will take a much more thorough look by zooming in to the per-OST traffic log, to discover their detailed I/O characteristics. This process serves two-fold purposes to *validate* the candidates' I/O intensity and for those validated, to collect I/O patterns relevant to subsequent I/O-aware scheduling. More specifically, AID collects (1) the application's aggregate I/O volume per I/O burst, (2) the I/O interval (average time between two adjacent I/O-bursts) and (3) average I/O throughput.

In addition, assisted by the per-OST traffic analysis, AID derives an applications' OST footprint, the number of OSTs it tends to use simultaneously. With n compute nodes, typical I/O-intensive applications may use independent I/O to write n files (n-to-n model), or collective I/O to write one (n-to-1) or m (n-to-m) files [6], [9]. Finding the OST footprint also serves two-fold purposes. First, it enables the I/O-aware scheduler to estimate how many OSTs an application uses, and assess the chance of two such applications stepping on each other's toes (though there is currently no way for the scheduler to force an application to use a certain group of OSTs). Second, pinpointing the subgroup of OSTs an application used allows our I/O characterization process to refine the I/O patterns collected, as I/O traffic from OSTs considered unused by this application can now be excluded.

A. Initial I/O Intensity Classification

From jobs to applications: I/O characteristics belong to *applications*, but are observed through *jobs*, each a particular execution of an application. The number of unique applications is typically much smaller than the number of jobs run per year. Meanwhile, the same application run by different users (domain scientists, system software experts, and software engineers) or run with different number of nodes may exhibit different I/O behavior. On the other hand, we observed that it is quite rare for a single user to incur very different I/O patterns running the same application using the same *node count* (the number of compute nodes used by a job).

Therefore, for I/O characterization, we define a *unique application* in the context of AID with the 3-tuple $\langle$*user name*, *job name*, *node count*$\rangle$. Here "job name" is a user-assigned string identifier included in the job script. This definition allows us to obtain 9998 unique applications from the 5-month log containing 181,969 jobs, resulting in 18.2 job runs per application on average during this period.

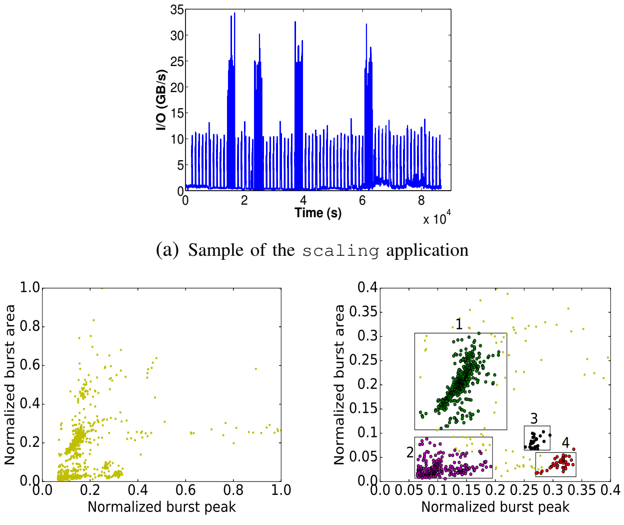
Candidate selection: Now we need to examine the aggregate I/O traffic log to nominate I/O-intensive application candidates. This is done by processing the samples of each application, obtained by intercepting the aggregate traffic log using its jobs' start/end times, in search for consistent and significant I/O activities. The key property utilized by AID is the periodicity of an application's I/O behavior. E.g., Figure 4(a) plots an original sample of a real application, named `scaling`, showing I/O bursts with clear periodical pattern. As mentioned earlier, existing techniques from IOSI [18] are adopted to identify individual I/O bursts from each sample.

Unlike IOSI, however, AID has to deal with the irregularity and inconsistency involved in classifying unknown real applications whose job runs may possess variable I/O behavior, aside from the noisy background from the aggregated I/O traffic log. The bursts found in a sample could belong to any of the concurrently running applications or even interactive user commands, and we have no knowledge to assume any application to possess I/O dominance in these samples. To solve this problem, we adopt a density-based clustering technique, OPTICS [4], by transforming each identified I/O burst for an application (all its samples combined) to a point in a 2-D space, using the burst height (peak aggregate I/O throughput) and area (total I/O volume) as x and y coordinate, respectively. AID then performs clustering of these points, aiming to identify groups of highly similar I/O bursts generated by potentially periodic I/O operations. We experimented with multiple widely-used clustering algorithms, including K-means [22] and DBSCAN [13], and finally selected OPTICS, which is very robust with noisy data and does not make any assumption on the number or shape of result clusters.

With 5 such samples of `scaling`, AID identifies a total of 1070 I/O bursts and mapped them to the aforementioned 2D space (Figure 4(b)). Figure 4(c) further displays the "zoomed-in" area with 4 result clusters identified by OPTICS.

AID then splits an original sample into n *sub-samples*, each containing the I/O bursts from one of the n identified clusters. To give an example, Figure 5 displays 4 sub-samples from the original sample shown in Figure 4(a), plotted using colors corresponding to the clustering result in Figure 4(c).

Its subsequent processing is based on the intuition that if a group of I/O bursts belongs to the application in question, such



(a) Sample of the `scaling` application

(b) Mapping `scaling` samples

(c) Clustering result of the samples

Fig. 4. Example of OPTICS clustering

I/O bursts need to (1) possess regular periodic pattern, and (2) appear in a significant portion of the original samples. By examining aggregate statistics such as I/O interval (time distance between adjacent I/O bursts), peak I/O throughput, and total I/O volume per-burst, we avoid the "sample alignment" and "full signature mining" tasks relying on strong assumptions regarding the existence and consistency of application I/O pattern. With attention instead focused on burst "size", "height", and "frequency", AID can handle variable execution lengths. Similarly, by requiring only a fraction of samples possessing common pattern (using a configurable threshold, set to 60% in our evaluation), coupled with differentiating the same base application's jobs submitted by different users, AID can handle inconsistent I/O patterns.

After such screening, among the I/O bursts in the 4 sub-samples shown in Figure 5, only one (cluster 1) is identified as a valid I/O pattern, as it is found to be regular, spanning a significant portion of the job execution, and present across over 60% of sub-samples. Note that cluster 4 (Figure 5(d)), though similarly regular looking, does not satisfy this standard and is considered from other applications. The other clusters also fail to meet these requirements. Finally, applications with such identifiable I/O pattern (at least one verified I/O burst cluster) are preliminarily considered an I/O-intensive candidate.

B. Candidate Application Validation

To reduce false positives, AID applies two validation techniques to all candidates, as discussed below.

Scope checking This is to guard against the case where there are long and frequent executions of a true I/O-intensive application, whose samples entirely "cover" certain other applications' shorter samples often enough. Those other applications will then have samples sharing the same I/O pattern. The solution is rather intuitive: for each qualifying sample, we look beyond its boundary, left and right, to check the correlation between the I/O pattern's existence and the application's execution. If a detected pattern is indeed incurred by a certain application, it should not be consistently observed before/after its job starts/completes running. Our AID prototype performs this checking by empirically examining 5 times of the detected I/O interval length, each way beyond the sample boundary.

Minimum support requirement Still, there could be relatively rare cases where a candidate happens to piggy-back on true I/O-intensive applications with similar job start and end times. This is more likely to happen when the false positive only has few samples. As mentioned earlier, AID is designed to be a self-learning system, scheduled to run at least daily to incrementally process new samples. Therefore, it maintains a separate watch list for "under-probation" candidates, who need to be validated with more samples. After bootstrapping the knowledge base, all new applications need to go through this probation period. Currently our AID prototype requires 5 minimum samples to have a candidate validated, which, together with the aforementioned scope checking, significantly reduces the chance of admitting a false positive.

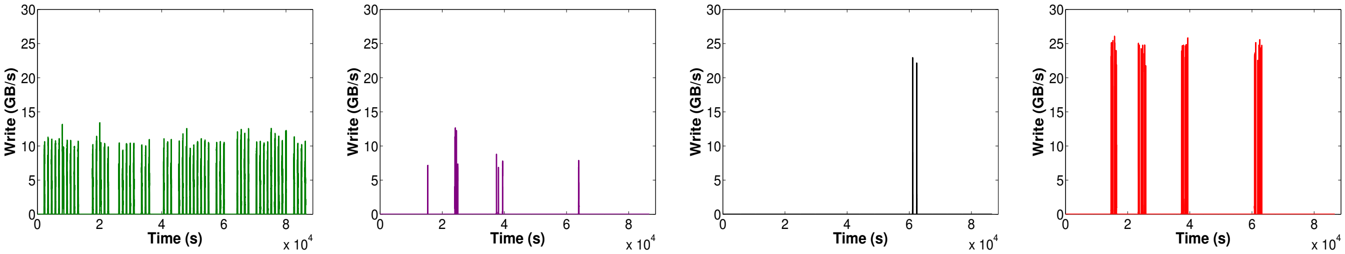
C. Application OST Utilization Analysis

With I/O-intensive applications identified and validated, AID performs another round of more detailed analysis. It now has the total I/O volume per I/O burst, the peak/average I/O throughput during I/O bursts, the I/O burst interval, and the computation-to-I/O time ratio. These features describe the *temporal distribution* of an application's I/O traffic and can be obtained by analyzing the aggregate server-side traffic log.

To understand an application's I/O behavior from the *spatial* aspect, AID mines its OST footprint, the number of OSTs it accesses during an I/O burst, as the final feature of its I/O pattern. Collecting and analyzing OST-level traffic logs is time consuming, as each sample (called *aggregate sample* hereafter) becomes 1008 $OST\ samples$, due to the lack of information on which OSTs were mapped to a job. Fortunately, we have dramatically reduced our scope of examination by identifying I/O-intensive candidates from thousands of applications.

OST footprint identification Our log analysis finds real-world applications consistent in OST footprint across I/O bursts, across both read and write operations. Therefore, AID assumes a constant OST footprint, κ, for each application. Each individual burst in an aggregate sample then comprises of κ OST bursts ($0 < \kappa \leq 1008$). For example, if an application writes collectively a single global shared file (n-to-1 model), with a file stripe width of 16, κ = 16. If the average OST throughput is 200 MB/s (with no other concurrent activities), the aggregated sample will contain I/O bursts with I/O throughput of around 3.2 GB/s.

After preparing OST samples for each aggregated sample, AID reuses the OPTICS clustering results described earlier. These aggregate bursts have already been certified as "regular and consistent" in the previous steps, hence giving strong hints for the search of similar-shaped bursts on individual OSTs. Suppose we have n aggregate samples from our target application, each bearing m aggregated bursts. For each aggregate



(a) Subsample of cluster 1 (b) Subsample of cluster 2 (c) Subsample of cluster 3 (d) Subsample of cluster 4

Fig. 5. Restructured sample based on clustering result

burst B_i in the aggregate sample S_j, AID scans *all 1008 per-OST samples* and counts $o_{i,j}$ corresponding single-OST bursts. Then the application's OST footprint κ is calculated as

$$\kappa = (\sum_{j=1}^{m} \sum_{i=1}^{n} o_{i,j})/mn \qquad (1)$$

This way, κ gives an estimated average number of OSTs participating in each I/O burst. We have evaluated such footprint estimation with both real applications and pseudo-applications, and found it to be effective (results in Section VI-B).

OST-footprint-enabled I/O pattern refinement Finally, we perform another round of fine tuning, by taking into account the AID-identified OST footprint. With this additional piece of information, we can pinpoint the subset of OSTs suspected to have participated in each I/O burst. This allows us to exclude I/O traffic from other OSTs from the aggregate sample, identified as background I/O noise from the target application's point of view. Though not affecting features such as I/O interval, such refinement improves the accuracy of per-burst I/O volume and average I/O throughput, as demonstrated by our experimental results in Fig. 8.

V. I/O-aware Job Scheduling

Based on the application I/O intensity classification and I/O pattern characterization results, AID explores I/O-aware scheduling. As discussed earlier, it is very hard to test-deploy such proof-of-concept prototypes on large, production supercomputers. Simulation study, on the other hand, cannot reflect the real dynamics of the mixed workload and shared resources of such large-scale systems. AID hence starts with a best-effort *recommendation system*, where it gives simple "run" or "delay" recommendations based on its self-learned knowledge *and* real-time system load information. We can then evaluate the validity and potential impact of such recommendations, by comparing what happens with dispatching the application run anyway, according to or despite AID's suggestion.

Another path is *spatial partitioning*, splitting OSTs across applications when possible. This can potentially be applied independently or jointly with the temporal staggering of jobs. However, its deployment requires significant modification of both the scheduler and the parallel file system, making it even harder for us to verify. That said, the current AID does consider the estimated per-application OST footprint in making scheduling recommendations.

A. Summarizing I/O-Related Information

First, AID needs to establish a global view of I/O requirements of ongoing applications as well as the real-time storage system status showing how busy individual OSTs are. More specifically, it gathers information from two aspects:

Current application I/O requirement Using its application knowledge base, AID can easily identify I/O-intensive applications running or waiting. It can further retrieve each candidate's I/O characteristics, to estimate the resource requirement for upcoming jobs, as well as to estimate the remaining execution time of jobs already running. Note that despite the effectiveness of its I/O workload auto-characterization, AID is a *best-effort* system that does not possess sufficient ground truth on its classification or I/O pattern mining results. Nor can it *guarantee* that the future will repeat the history.

Real-time system I/O load level Meanwhile, AID continuously monitors the current I/O system load level by maintaining an OST-level *I/O load table (IOLT)*, updated by querying the same server-side per-OST traffic log data, once every 5 minutes in our prototype. The IOLT maintains a load level histogram for each OST during the last four 5-minute windows, in terms of its logged I/O throughput. In other words, it keeps track of a 20-minute sliding window showing detailed recent history of per-OST load level. AID divides the per-OST throughput range $[0, T_p]$ (T_p being the peak throughput) into uniform throughput level bands (with width of 30MB/s in our prototype). For each such band (*e.g.*, $[60MB/s, 90MB/s]$) per 5-minute window, AID stores in IOLT the fraction of time that each OST has load level within this interval, say 20%.

The real-time system load information allows AID to supplement its knowledge base with actual run-time system status, making it aware of both the "demand" and "supply" sides of the shared I/O resources. Also, the real-time system load data compensate AID's lack of knowledge on newer applications that do not have (sufficient) samples or interactive user/administrator activities that bypass batch scheduling.

B. I/O-aware Scheduling

Finally, AID puts together everything it knows to make I/O-aware scheduling recommendations: whether A should be dispatched now ("run") or later ("delay"). AID does this by calculating a numeric *OST load score* ("*score*" for short), and making job admission decisions considering A's estimated OST-footprint and I/O traffic. Below we describe the major steps involved in this decision making process.

OST load score calculation This step takes the IOLT as input and calculates for each OST the load score s:

$$s = \sum_{i=1}^{n} w_i(f_i + \alpha) \tag{2}$$

Here n is the total number of the aforementioned per-OST throughput level bands and f_i is the fraction of time this OST stayed within the ith throughput band, based on the recent history from the IOLT. This is to roughly measure the *chance* and *degree* that A is expected to endure I/O contention with its immediate dispatch, by examining both the frequency and intensity of existing I/O activities, instead of simply relying on the average/peak/minimum throughput. On top of f_i, we make an additional adjustment, α, leveraging the I/O patterns mined by AID, for ongoing applications that has just started or is about to complete (based on application job history and the maximum execution time submitted to the scheduler and available at real time to AID). Basically, we use their I/O pattern in the knowledge base to add/subtract I/O throughput intervals for newborn/dying jobs. Since we do not have the mapping from application to a particular set of OSTs, we make such adjustment at the top K idle/busy OSTs, where K is the estimated OST footprint of the newborn/dying application.

However, with complex resource sharing behaviors and without detailed ground truth, we cannot fully understand/predict the impact the background load has on A's execution on the same OST. To this end, in Equation 2 we add a weight, w_i, to the corresponding band. The weight values are in turn to be learned in a black-box style by real-system I/O interference measurement, collected in a 2-month period on Titan, during which we submitted small training jobs with known I/O patterns, to measure their I/O performance behavior under different OST load levels. More specifically, we simply used the I/O time of a training job normalized to the measured shortest time in all trials as s in Equation 2, making w the only unknown. With a large number of training data points, we solve the w values using an over-determined system [5].

Application-specific load threshold calculation Taking a similar approach as in weight calculation above, AID also observes the impact of overall OST load (in terms of average s score over all OSTs used) on a target application by measuring such correlation between system load and training job's I/O performance. Here it maintains a 2-D data structure, partitioning the per-application I/O pattern into coarse intervals using two parameter values: the average I/O throughput per-burst, and the I/O-to-computation time ratio. Each "cell" in this 2-D table saves training data points within the corresponding parameter range, recording the measured average OST load upon dispatch (again mined from history logs) and normalized I/O performance. Therefore, given a known I/O-intensive application A and its I/O pattern retrieved from the AID knowledge base, plus a configurable performance impact factor (*e.g.*, a factor of 1.2 means that 20% longer I/O time can be accepted), we can utilize the pre-computed correlation and derive the threshold average OST load level L.

I/O-pattern-aware OST screening With the per-OST load score s calculated based on recent load history, and the average OST load level L calculated based on A's known I/O pattern, AID checks whether there are enough OSTs with projected I/O load low enough to accommodate A now.

This is done by obtaining A's OST footprint m from the knowledge base, and examine the m OSTs with the lowest load (by s value) in each file system partition. If the average load of such "least busy" OSTs is under A's application-dependent load threshold L, then AID makes the "run" recommendation, encouraging the immediate execution of A. Otherwise, it makes the "delay" recommendation to hold A in the batch queue until next (event-prompted) evaluation point.

VI. Experimental Evaluation

We implemented a proof-of-concept prototype of AID, in around 3200 lines of Python code. The tool itself has small overhead. More specifically, it took no more than 9 hours to bootstrap the knowledge-base using 6 months' logs, around 3 minutes for its daily knowledge-base update using new logs, and around 1 or 2 seconds for making an online scheduling recommendation.

Our evaluation aims to verify several main hypotheses: (1) AID can successfully identify I/O-intensive applications without apriori information; (2) AID can identify I/O-intensive applications' OST footprint with reasonable accuracy; and (3) I/O-aware job scheduling based on automatically derived per-job I/O behavior can effectively reduce I/O contention.

Note that AID analyzes real application data, but we have to rely on (unknown) users' reply to our query to obtain some "ground truth" for the majority of user jobs on Titan. Therefore, we also generated our own pseudo-applications, again with IOR (more details in Section VI-B). These applications possess typical real I/O patterns observed on Spider, and have been submitted repeatedly during several weeks.

A. I/O Intensity Classification

Name	Value
Total number of logged jobs	181,969
Real applications without 3-tuple identification	460
Real applications with 3-tuple identification	9,998
Initial I/O-intensive candidates	95
Candidates passing scope checking	67
Candidates passing min support requirement	42
User-verified candidates	8

TABLE I
Real application classification results

Our classification evaluation mainly focuses on checking against false positives, as our "I/O-intensive" definition requires "observable" I/O patterns and AID is rather confident when an application cannot even make its watchlist.

Real applications We fed AID with the aforementioned 5-month Titan I/O traffic and job logs. Table I summarizes major statistics information regarding logged jobs.

AID obtained 95 preliminary I/O-intensive candidates, and with its own validation using scope checking and minimum support requirement it cut the shortlist to 42 candidates. We

ID	Node	Time(m)	OST	App. Domain
1	8192	1440	64	Geo-sciences
2	250	6-60	1008	Combustion
3	2048	30-185	1008	Astrophysics
4	1760	720	180	Combustion
5	1024	110-230	1008	Systems research
6	200	30-190	1008	Combustion
7	1008	13-17	1008	Computer Science
8	16388	43-310	800	Environmental

TABLE II
USER-VERIFIED I/O-INTENSIVE APPLICATIONS

hoped to verify the findings with feedback from the application owners, however contacting Titan users has to comply with center policy and is non-trivial. In the end, we obtained approval to contact 16 candidates, mostly submitted by local users. We contacted them by email and received responses from 8, all confirming of the I/O-intensive classification. Table II briefly describes these user-verified candidates.

Pseudo-applications First, Table III lists the characteristics of our own IOR pseudo-applications, whose trial runs (submitted at diverse times of the day) consumed around 834,682 node-hours on Titan. We specifically varied the number of iterations (computation phase plus I/O phase), to evaluate AID's capability of handling variable-length executions of the same application. Also, we configured IOR_H with smaller output size per compute node and the *n-to-1* model to generate rather low I/O throughput. As expected, AID correctly identified all 7 I/O-intensive pseudo-applications and did not admit IOR_H as a candidate. Moreover, all of our pseudo-applications are run with the real jobs on Titan, making them among the 42 AID-identified I/O-intensive applications.

ID	# Nodes	I/O interval	# iter.	Burst vol.
IOR_A	256	300s	6-8	1024GB
IOR_B	512	250s	6-10	2048GB
IOR_C	128	450s	4-6	1024GB
IOR_D	1024	600s	4-5	2048GB
IOR_E	128	100s	10-20	1024GB
IOR_F	128	80s	15-25	1024GB
IOR_G	256	600s	12-20	2048GB
IOR_H	64	500s	4-6	64GB

TABLE III
PSEUDO-APPLICATION CONFIGURATIONS

B. I/O Pattern Identification

	Burst volume	Interval	Throughput	# OST
AID	34TB	350s	150GB/s	960
Actual	32.4TB	380s	184GB/s	963

TABLE IV
APPLICATION 8, AID VS. GROUND TRUTH

Real applications We evaluate AID's capability of mining detailed I/O characteristics using the 8 verified real-world applications. Table II confirms that I/O-intensive applications are indeed run in diverse scales and lengths (producing large ranges of node count and execution time distribution).

In addition to confirming I/O intensity, their users kindly filled our email questionnaire on I/O behavior and settings. Figure 6 shows the side-by-side comparison between AID-extracted and user-supplied I/O patterns. We examine four key features, namely per-burst I/O volume, I/O interval, average I/O throughput during bursts, and OST footprint. We choose not to normalize the results to show the actual scale and distribution of such pattern features in real applications. Results for Application 8 are given separately in Table IV, due to its exceptionally large I/O volume.

These results show that AID achieves high accuracy in automatically discovering application-specific I/O characteristics, with errors likely due to noises. Actually, AID estimated I/O volume can sometimes be smaller than the true application volume, indicating that we might have excluded I/O traffic from the target application. However, such accuracy suffices for I/O-aware scheduling and workload study purposes.

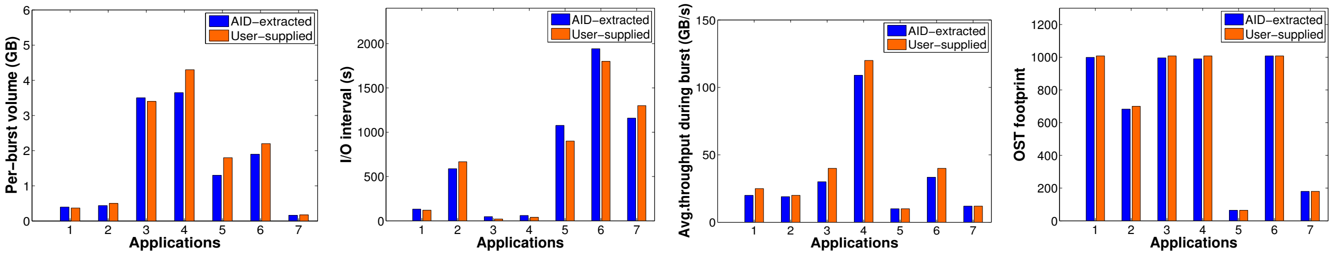
One interesting side discovery here is that the majority of observable I/O-intensive applications have rather large OST footprint, as currently this is still the parallel I/O model that delivers the highest aggregate throughput by avoiding synchronization overhead (even when using the same number of OSTs). More efficient *n-to-m* parallel I/O would allow the applications to obtain high throughput while leaving the system more flexibility in I/O-aware scheduling.

Pseudo-applications Unlike with real applications, we possess all ground truth on our IOR pseudo-applications. We designed them to portray the diverse HPC I/O behavior, with contrasting node counts, per-burst I/O volumes, and I/O intervals. Most importantly, they adopt different common HPC file access models (*n-to-1*, *n-to-m*, and *n-to-n*), resulting in different OST footprints. To match the behavior observed in real applications, we intentionally added variability in job behavior (while maintaining the base I/O pattern), by changing the number of computation-I/O iterations, hence producing different sample lengths. In addition, we simulate the "I/O-off" runs by randomly adding 1 - 5 job runs without I/O. The output files use the default Spider setting: stripe count of 4 and stripe size of 4MB. Figure 7 confirms that AID achieves similarly good accuracy in deriving the application I/O patterns.

Fig 8 demonstrates the effect of I/O pattern refinement on one of the IOR_B samples using the OST footprint results, as described in Section IV-C. The left figure shows I/O bursts before having OST footprint information and the right one after. The I/O bursts after refinement are visibly more clarified and less noisy. As marked on the top of the figures, the OST-footprint knowledge (verified as quite accurate by previous results) helps AID trim the total I/O volume of this application from 53.4TB to 38.5TB within this particular sample, by excluding I/O traffic from non-participating OSTs.

C. I/O-aware Job Scheduling

Finally, we evaluate the effectiveness of AID's I/O-aware scheduling recommendation. In this set of experiments, we issued groups of pseudo-applications to create varying levels of inter-job I/O contention as well as system I/O load. These experiments were conducted on the Titan production system, where we had no control on actual job concurrency. Titan



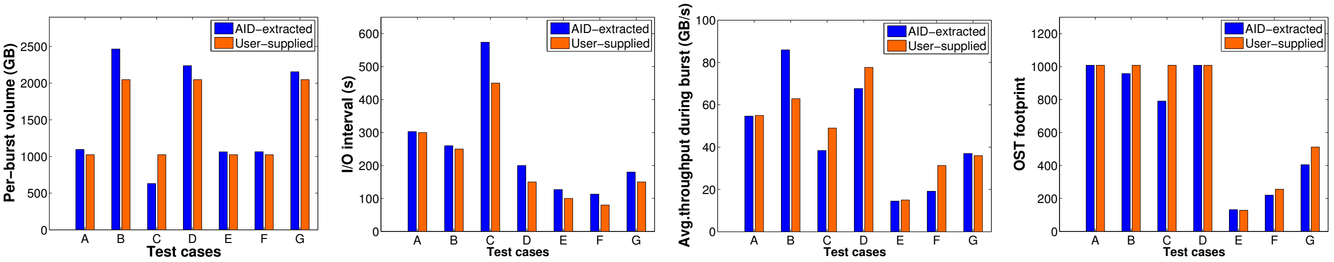
(a) I/O volume per I/O burst

(b) I/O interval

(c) Average I/O throughput

(d) Number of OSTs

Fig. 6. Real Titan application I/O characterization accuracy



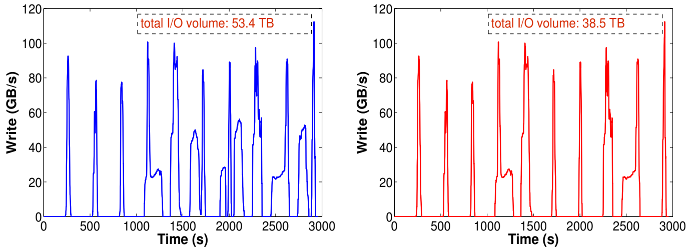
(a) Volume per I/O burst

(b) I/O interval

(c) Average I/O throughput during burst

(d) OST footprint

Fig. 7. IOR pseudo-application I/O characterization accuracy



(a) Before OST analysis

(b) After OST analysis

Fig. 8. I/O bursts before/after OST footprint analysis

job logs show that at any given time point during these tests, 33-68 jobs ran together with our ”target pseudo-application”, of which 1-7 are our other pseudo applications. We analyze ongoing applications' I/O patterns and calculate the per-OST load scores, as discussed in Section V, which leads to either the “run” or “delay” recommendation right before the target pseudo-application's execution.

Note that *we let the jobs run under the ”delay” recommendation anyway*, and check whether the ”delay” suggestion does correlate with worse contention. To simulate the scenario with real I/O-aware scheduling enabled, after the target application starts we check and suspend other queued pseudo-applications, to isolate the evaluation of individual scheduling decisions.

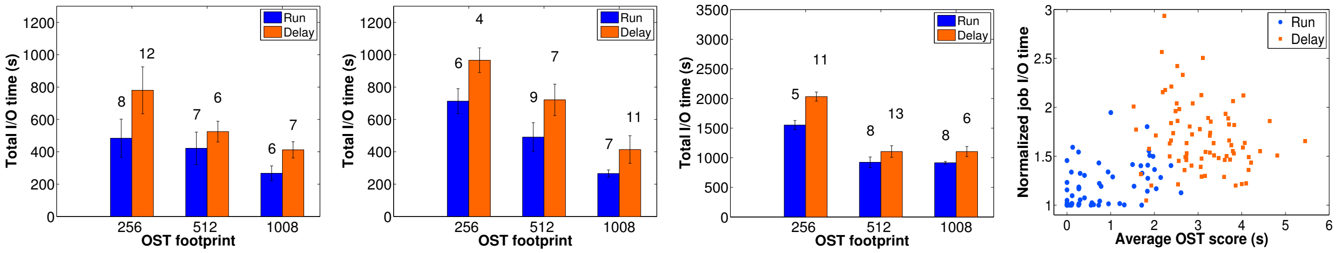
Here we used another set of IOR pseudo-applications (different from those in Table III), all using the *n-to-m* model, to control the OST footprint size (256, 512, and 1008). The acceptable performance impact factor is set to 1.5 (50% slowdown). Figure 9 gives the results, with pairs of bars showing the average per-job I/O time, plus variance in standard deviation. The number above each bar indicates the number of trials. We had to run many jobs to get at least 5 trials receiving either recommendation, as we cannot control whether an individual trial will receive a “run” or a “delay” order. As expected, the runs *started despite the “delay” recommendation* do spend considerably more time on I/O and often have larger I/O time variances compared to those with the “run” blessing, as seen in Figures 9(a) - 9(c).

Figure 9(d) plots the 2-D distribution of all trial data points, in average OST score (x) and total I/O time normalized to the shortest measurement (y). It clearly shows that the “run” data points (blue dots) have better and more consistent performance than the “delay” ones (orange squares). More specifically, the “delay” data points have an I/O performance impact factor (slowdown from the shortest I/O time measurement) of 1.69 on average and up to 2.93. The “run” data points, in contrast, have 1.21 on average and up to 1.92.

Several of the “run” data points do get over the 1.5 impact factor threshold and cause larger variances in the “run” bars in Figures 9(a) - 9(c). After all, the experiments are done on a large production system where we are not really scheduling applications: though we can “hold” other pseudo-applications, real I/O-intensive jobs do not go through AID's approval and may start after the pseudo-application's launch. Therefore we expect AID's advantage to be more significant with fully deployed I/O-aware schedulers.

VII. Related Work

Resource-aware job scheduling I/O contention has been recognized as an important problem for current and future large-scale HPC platforms [7], [11], [14], [16], [20], [28], [39]. Two studies have proposed platform-level, I/O-aware job scheduling for reducing inter-job interference. Dorier et al. proposed CALCioM [11], which dynamically selects appropriate scheduling policies, coordinating applications' I/O strategy via inter-application communication. Applications on the same compute platform are allowed to communicate and coordinate



(a) Total I/O time, IOR_X

(b) Total I/O time, IOR_Y

(c) Total I/O time, IOR_Z

(d) Normalized I/O time, IOR_{X-Z}

Fig. 9. Performance advantage of following scheduling recommendation

their I/O strategy with each other to avoid interference. Zhou et al. [38] and Thapaliya et.al [31] proposed solutions coordinating/admitting I/O accesses to reduce I/O interference, but both approaches require parallel file system modifications. Gainaru et al. proposed a global scheduler [14], which prioritizes I/O operations across applications based on applications' past behavior and system characteristics to reduce I/O congestion. Compared to these systems, AID is much more lightweight, and does not require inter-application communication or system-level modification/overhead to schedule I/O operations. In addition, the above existing global schedulers/coordinators only moderate operations from already scheduled applications, in contrast to AID's proactive I/O-aware scheduling, especially between applications with large OST footprints.

A few studies have proposed resource-aware job scheduling to alleviate inter-job resource contention [28], [39], e.g., by considering jobs' communication patterns. Wang et al. [33], developed a library, *libPIO*, that monitors resource usage at the I/O routers, OSSes, OSTs, and the SION InfiniBand Switches, and based on the load factor allocated OSTs to specific I/O clients. Other systems have explored application-level I/O aware scheduling. Li et al. [17] proposed ASCAR, a storage traffic management framework for improving bandwidth utilization and reducing performance variance. ASCAR focuses on QoS management between co-running jobs, instead of scheduling high-risk jobs. Novakovic et al. [25] presented DeepDive, which transparently identifies and manages interference on cloud services. Lofstead et al. [20] proposed an adaptive I/O approach that groups the processes of a running application, and directs their output to particular storage targets, with inter-group coordination. Zhang et al. [37] proposed IOrchestrator, which creates extra processes to execute I/O-intensive application code and retrieve information on future I/O requests for scheduling. These techniques are complementary to our approach. Meanwhile, AID's global scheduling aims to stagger the relatively small number of high-impact I/O-intensive applications away from each other. It strives to reduce the labor and performance cost of application-level I/O scheduling, as well as the potential side-effect of uncoordinated scheduling optimization by individual applications.

I/O characterization A number of I/O tracing tools have been developed for general-purpose client-side instrumentation, profiling, and tracing of I/O activity, including LANL-Trace [1], Darshan [10], HPCT-IO [30], RIOT I/O [36], ScalaIOTrace [32], TRACE [23], Omnisc'IO [12], and IPM [35]. However, existing tools suffer from multiple well-known limitations, such as system overhead (including generating additional I/O traffic), installation/linking requirements, and voluntary participation. Very recently, researchers performed a comprehensive application I/O characteristics study from several production supercomputer systems [21]. The authors successfully collected profiling data from a large fraction of applications using Darshan and their results provided valuable support on application behavior for AID's design decisions. Meanwhile, AID utilizes existing server-side monitoring and log data (collected with near-zero overhead), and can provide additional application I/O characteristics data to HPC storage/application developers with no user involvement or application modification.

VIII. Conclusion

In this paper, we present AID, a mechanism that *mines* application-specific I/O patterns from existing supercomputer server-side I/O traffic logs and batch job history jobs, *without any additional tracing, profiling, or user-provided information*. We verified the effectiveness of AID using both user feedback (on real-world HPC application unknown to us) and our own pseudo-applications on a state-of-the-art supercomputer. We further enabled AID to make I/O-aware scheduling recommendations, and confirmed with experiments on the same supercomputer that such recommendations can produce significantly lower I/O time and smaller I/O performance variance.

This work demonstrates that in large, complex, and highly dynamic shared environments, where detailed tracing/profiling is often intrusive and costly, we can still learn a lot about unknown applications just by examining low-overhead, coarse-granule system logs that have been routinely collected. The key observation here is that resource-heavy applications tend to have consistent behavior to be noticed, and the future, to a large extent, does repeat history.

Acknowledgement

We thank the reviewers for constructive comments that have significantly improved the paper. This work was supported in part by the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is managed by UT Battelle, LLC for the U.S. DOE (under the contract No. DE-AC05-00OR22725). This work was also supported in part by the NSF grants CCF-1318564.

REFERENCES

[1] Los Alamos National Laboratory open-source LANL-Trace, http://institute.lanl.gov/data/tdata.

[2] OLCF Policy Guide, https://www.olcf.ornl.gov/support/system-user-guides/olcf-policy-guide/.

[3] Titan, http://www.olcf.ornl.gov/titan/.

[4] Mihael Ankerst, Markus M Breunig, Hans-Peter Kriegel, and Jörg Sander. Optics: ordering points to identify the clustering structure. In *ACM Sigmod Record*. ACM, 1999.

[5] I Barroda and FDK Roberts. Solution of an overdetermined system of equations in the l1 norm [f4]. *Communications of the ACM*, 1974.

[6] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009.

[7] Rupak Biswas, MJ Aftosmis, Cetin Kiris, and Bo-Wen Shen. Petascale computing: Impact on future nasa missions. *Petascale computing: architectures and algorithms*, 2007.

[8] Peter J Braam and Rumi Zahir. Lustre: A scalable, high performance file system. *Cluster File Systems, Inc*, 2002.

[9] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Transactions on Storage (TOS)*, 2011.

[10] Philip H. Carns, Robert Latham, Robert B. Ross, Kamil Iskra, Samuel Lang, and Katherine Riley. 24/7 Characterization of petascale I/O workloads. In *Proceedings of the First Workshop on Interfaces and Abstractions for Scientific Data Storage (IASDS'09)*, 2009.

[11] Matthieu Dorier, Gabriel Antoniu, Robert Ross, Dries Kimpe, and Shadi Ibrahim. CALCioM: Mitigating I/O interference in HPC systems through cross-application coordination. In *Parallel and Distributed Processing Symposium*, 2014.

[12] Matthieu Dorier, Shadi Ibrahim, Gabriel Antoniu, and Rob Ross. Omnisc'IO: a grammar-based approach to spatial and temporal i/o patterns prediction. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014.

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[14] A. Gainaru, G. Aupy, A. Benoit, F. Cappello, Yves Robert, and M. Snir. Scheduling the of HPC applications under congestion. In *IPDPS*, 2015.

[15] Yasuhiko Kanemasa, Qingyang Wang, Jack Li, Masazumi Matsubara, and Calton Pu. Revisiting performance interference among consolidated n-tier applications: Sharing is better than isolation. In *Services Computing (SCC), IEEE International Conference*, 2013.

[16] Youngjae Kim, Scott Atchley, Geoffroy R Vallée, and Galen M Shipman. LADS: Optimizing data transfers using layout-aware data scheduling. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, 2015.

[17] Yan Li, Xiaoyuan Lu, Ethan L Miller, and Darrell DE Long. Ascar: Automating contention management for high-performance storage systems. In *Mass Storage Systems and Technologies (MSST'15)*, 2015.

[18] Yang Liu, Raghul Gunasekaran, Xiaosong Ma, and Sudharshan S Vazhkudai. Automatic identification of application I/O signatures from noisy server-side traces. In *Proceedings of the 12th USENIX conference on File and Storage Technologies*, 2014.

[19] Jay Lofstead and Robert Ross. Insights for exascale IO APIs from building a petascale IO API. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 2013.

[20] Jay Lofstead, Fang Zheng, Qing Liu, Scott Klasky, Ron Oldfield, Todd Kordenbrock, Karsten Schwan, and Matthew Wolf. Managing variability in the IO performance of petascale storage systems. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010.

[21] Huong Luu, Marianne Winslett, William Gropp, Robert Ross, Philip Carns, Kevin Harms, Mr Prabhat, Suren Byna, and Yushu Yao. A multiplatform study of I/O behavior on petascale supercomputers. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015.

[22] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, 1967.

[23] Michael P Mesnier, Matthew Wachs, Raja R Simbasivan, Julio Lopez, James Hendricks, Gregory R Ganger, and David R O'Hallaron. //trace: Parallel trace replay with approximate causal events. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*, 2007.

[24] Ross Miller, Jason Hill, David A. Dillow, Raghul Gunasekaran, Shipman Galen, and Don Maxwell. Monitoring tools for large scale systems. In *Proceedings of the Cray User Group (CUG'10)*, 2010.

[25] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, 2013.

[26] Sarp Oral, David A Dillow, Douglas Fuller, Jason Hill, Dustin Leverman, Sudharshan S Vazhkudai, Feiyi Wang, Youngjae Kim, James Rogers, James Simmons, et al. OLCF's 1 TB/s, next-generation lustre file system. In *Proceedings of Cray User Group Conference (CUG'13)*, 2013.

[27] Sarp Oral, James Simmons, Jason Hill, Dustin Leverman, Feiyi Wang, Matt Ezell, Ross Miller, Douglas Fuller, Raghul Gunasekaran, Youngjae Kim, et al. Best practices and lessons learned from deploying and operating large-scale data-centric parallel file systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014.

[28] Jordà Polo, Claris Castillo, David Carrera, Yolanda Becerra, Ian Whalley, Malgorzata Steinder, Jordi Torres, and Eduard Ayguadé. Resource-aware adaptive scheduling for mapreduce clusters. In *Middleware 2011*, pages 187–207. Springer, 2011.

[29] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. Who is your neighbor: Net I/O performance interference in virtualized clouds. *Services Computing*, 2013.

[30] Seetharami Seelam, I-Hsin Chung, Ding-Yong Hong, Hui-Fang Wen, and Hao Yu. Early experiences in application level I/O tracing on Blue Gene systems. In *Proceedings of the International Parallel Distributed Processing Symposium (IPDPS'08)*, 2008.

[31] Sagar Thapaliya, Purushotham Bangalore, Jay Lofstead, Kathrn Mohror, and Adam Moody. IO-cop: Managing concurrent accesses to shared parallel file system. In *Parallel Processing Workshops (ICCPW), 43rd International Conference on*. IEEE, 2014.

[32] Karthik Vijayakumar, Frank Mueller, Xiaosong Ma, and Philip C Roth. Scalable I/O tracing and analysis. In *Proceedings of the 4th Annual Workshop on Petascale Data Storage*. ACM, 2009.

[33] Feiyi Wang, Sarp Oral, Saurabh Gupta, Devesh Tiwari, and Sudharshan Vazhkudai. Improving Large-Scale Storage System Performance via Topology-aware and Balanced Data Placement. In *The 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014.

[34] Feng Wang, Qin Xin, Bo Hong, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Tyce T. Mclarty. File system workload analysis for large scale scientific computing applications. In *Proceedings of the IEEE 21th Symposium on Mass Storage Systems and Technologies (MSST'04)*, 2004.

[35] Nicholas J Wright, Wayne Pfeiffer, and Allan Snavely. Characterizing parallel scaling of scientific applications using ipm. In *The 10th LCI International Conference on High-Performance Clustered Computing*, 2009.

[36] Steven A Wright, Simon D Hammond, Simon J Pennycook, Robert F Bird, JA Herdman, Ian Miller, A Vadgama, Abhir Bhalerao, and Stephen A Jarvis. Parallel file system analysis through application I/O tracing. *The Computer Journal*, 56(2):141–155, 2013.

[37] Xuechen Zhang, Kei Davis, and Song Jiang. IOrchestrator: improving the performance of multi-node I/O systems via inter-server coordination. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[38] Zhou Zhou, Xu Yang, Dongfang Zhao, Paul Rich, Wei Tang, Jia Wang, and Zhiling Lan. I/O-aware batch scheduling for petascale computing systems. In *Cluster Computing (CLUSTER)*. IEEE, 2015.

[39] Hongbo Zou, Xian-He Sun, Siyuan Ma, and Xi Duan. A source-aware interrupt scheduling for modern parallel I/O systems. In *Parallel & Distributed Processing Symposium (IPDPS'12)*, 2012.